TENTAMEN

Programmeren 1

vakcode: 192135000
datum: 10 april 2013
tijd: 8:45-12:15

Algemeen

- Bij dit tentamen mag gebruik worden gemaakt van het boek van Niño en Hosch, de Programmeren 1 handleiding en een kopie van de collegesheets zonder aantekeningen.
- Dit tentamen bestaat uit 4 opgaven, waarvoor in totaal 105 punten (100 punten + 5 bonus punten) behaald kunnen worden. Het minimale aantal punten per opgave bedraagt 0. Het cijfer wordt berekend als (aantal punten)/10, afgerond tot een geheel getal.
- Bij de opdrachten in dit tentamen hoeven *géén* pre- en postcondities te worden gegeven, tenzij *expliciet* anders vermeld. Neem wel commentaar op waar dat nuttig is voor het begrijpen van de oplossing.

Opgave 1 (15 punten)

De resolutie van een beeldscherm wordt tot uitdrukking gebracht in een tweetal getallen: het aantal punten langs de x-as en het aantal punten langs de y-as. Typische voorbeelden zijn: 640×480 en 1280×1024.

Geef drie verschillende manieren om beeldscherm-resolutie door middel van een (bestaand of zelf te definiëren) type in Java te representeren. Ga er van uit dat de hoogste resolutie die hoeft te worden weergegeven 65536×65536 (d.w.z., $2^{16} \times 2^{16}$) bedraagt. Geef *voor elk van deze manieren*:

- Het principe van de representatie, geïllustreerd met de representatie van de boven genoemde voorbeeld-resoluties.
- Een methode `<resolutie> newResolutie(int, int)` die, gegeven een x- en y-coordinaat, een nieuwe resolutie oplevert, methoden `int getX(<resolutie>)` en `int getY(<resolutie>)` die de x- en y-coordinaat van een gegeven resolutie opleveren en methode `boolean equalResoluties (<resolutie>, <resolutie>)` die de waarde **`true`** oplevert als de resoluties dezelfde zijn (en **`false`** als het niet zo is) . Hierin is `<resolutie>` het gekozen type.
- De voor- en nadelen van deze representatie.

Opgave 2 (25 punten)

De interface `Set<E>` in de standaard Java klassenbibliotheek kent o.a. de volgende methoden (teskt uit de Java API):

- `boolean add(E o)`: Adds the specified element to this set if it is not already present. Returns `true` if element is added.
- `boolean addAll(Collection<? extends E> c)`: Adds all of the elements in the specified collection to this set if they're not already present. Returns `true` if any element is added.
- `void clear()`: Removes all of the elements from this set.
- `boolean contains(Object o)`: Returns `true` if this set contains the specified element.

- `boolean isEmpty()`: Returns `true` if this set contains no elements.
- `boolean remove(Object obj)`: Removes the specified element from this set if it is present. Returns `true` if anything was removed.
- `int size()`: Returns the number of elements in this set (its cardinality).
- `Object[] toArray()`: Returns an array containing all of the elements in this set.

Een standaard-implementatie van `Set<E>` is `HashSet<E>` (met lege constructor). Hieronder een voorbeeld-programma dat gebruik maakt van `Set<E>`:

```
1   Set<Object> voorbeeld = new HashSet<Object>();
2   voorbeeld.add("Een");
3   voorbeeld.add(new Integer(2));
4   voorbeeld.add("Drie");
5   System.out.println(voorbeeld.contains(new String("Een")));
6   System.out.println(voorbeeld.contains(new Integer(1)));
7   Object[] elementen = voorbeeld.toArray();
8   Object elem0 = elementen[0];
9   System.out.println(elem0);
10  voorbeeld.remove(elem0);
11  System.out.println(voorbeeld.contains(elem0));
```

Dit programma levert de volgende uitvoer op:

```
true
false
2
false
```

Beantwoord de volgende vragen, *met voldoende toelichting op uw antwoord (5 punten per vraag voor een correcte antwoord met juiste toelichting)*:

1. Regel 1 declareert een variabele `voorbeeld` van type `Set<Object>` maar initialiseert hem met type `HashSet<Object>`. Waarom klopt dit wel? Wat is het voordeel van het declareren van de variabele `voorbeeld` met type `Set<Object>`?
2. De methode `contains(Object)` test of de parameterwaarde gelijk is aan (een van) de elementen in de verzameling. Gebeurt dat op basis van `==` of van `equals(Object)`? Het voorbeeldprogrammastuk en bijbehorende uitvoer geven indicaties van het juiste antwoord.
3. Welke objecten zitten in `elementen` na afloop van dit programmastuk?
4. Waarom is de waarde van `elem0` gelijk aan `2`?
5. Verandert de uitvoer van het programma als we regel 2 dupliceren, d.w.z. het element `"Een"` twee keer toevoegen?

Opgave 3 (30 punten)

In deze en de volgende opgave specificeren en implementeren we (een gedeelte van) een eenvoudige makelaarsapplicatie. Een makelaar heeft panden in de verkoop en hij kan zelf bij andere makelaars panden aankopen. Er zijn verschillende soorten panden, zoals, bijvoorbeeld, appartementen, villa's, tussenwoningen en hoekwoningen. In de makelaarsapplicatie hebben al deze panden één abstracte superklasse, de klasse `Pand`. Een pand kan maar bij één makelaar tegelijk in de verkoop zijn. Als een makelaar een pand in de verkoop heeft, dan kunnen andere makelaars zich bij de makelaar aanmelden als geïnteresseerden voor dat pand. Een verkopende makelaar kan maar met één geïnteresseerde makelaar tegelijk in onderhandeling zijn. De volgorde van aanmelden bij de verkopende makelaar is hierbij belangrijk: wie-het-eerst-komt, die-het-eerst-maalt. Als de verkopende makelaar niet met de eerste geïnteresseerde makelaar tot overeenstemming

kan komen, dan is de tweede makelaar aan de beurt. Een makelaar wordt gerepresenteerd door de klasse `Makelaar`.

Wanneer een pand in de verkoop komt, dan stellen de eigenaar en de verkopende makelaar samen een vraagprijs op; dit is het bedrag waarvoor ze het pand willen verkopen. Tijdens de onderhandeling doet de geïnteresseerde makelaar een tegenbod, waarvoor hij het pand wel zou willen kopen. Als de verkopende partij het tegenbod te laag vindt, maar wel verder wil onderhandelen kan hij de vraagprijs verlagen. Dit 'afdingen' gaat net zolang door totdat de verkopende en kopende partij het over de prijs eens zijn. Maar het komt natuurlijk ook voor dat de beide partijen niet tot overeenstemming kunnen komen. Hieronder volgt allereerst de partiële definitie van de klasse `Makelaar`.

```
public class Makelaar {
  private String naam;        // naam van de Makelaar
  private List<Pand> tekoop; // panden in de verkoop
  public Makelaar(String naam) {
    this.naam = naam;
    this.tekoop = new ArrayList<Pand>();
  }
  public String getNaam() { return naam; }
  public List<Pand> getPanden() { return tekoop; }
  /**
   * Bepaalt de gemiddelde vraagprijs van de panden die deze
   * Makelaar in de verkoop heeft.
   * @require ! this.getPanden().isEmpty()
   * @ensure result == gemiddelde vraagprijs van de
   *                   panden in this.getPanden()
   */
  public double gemiddeldeVraagPrijs() {
    // Te programmeren
  }
  /**
   * Breng een bod van bedrag euros uit op een pand.
   * @require pand != null && bedrag >= 0.0
   * @ensure als !this.getPanden().contains(pand)
   *              && pand.getVerkoper() == this
   *              && pand.getLaatsteBod() == bedrag
   *         dan result
   *         anders ! result
   * @param pand het Pand-object waarop een bod wordt gedaan.
   * @param bedrag het bod dat wordt uitgebracht
   */
  public boolean doeBod(Pand pand, double bedrag) {
    // Te programmeren
  }
}
```

Zoals u kunt zien worden in de klasse `Makelaar` alleen de naam van de makelaar en de lijst van `Pand`-objecten die de makelaar in de verkoop heeft bijgehouden. Hieronder volgt de eveneens partiële definitie van de abstracte klasse `Pand`.

```
public abstract class Pand {
  private String adres;
  private String postcode;
  private Date tekoopPer;
  private double vraagprijs;
  private double laatsteBod;
  private Makelaar verkoper;
  private List<Makelaar> geinteresseerden;
```

```
// Constructor
protected Pand(String adres, String postcode, Date tekoopPer,
    double vraagprijs, Makelaar verkoper) {
  this.adres = adres;
  this.postcode = postcode;
  this.tekoopPer = tekoopPer;
  this.vraagprijs = vraagprijs;
  this.verkoper = verkoper;
  this.laatsteBod = 0.0;
  this.geinteresseerden = new ArrayList<Makelaar>();
}
// De gebruikelijke get-queries.
public String getAdres() { return adres; }
public String getPostcode() { return postcode; }
public Date getTeKoopDatum() { return tekoopPer; }
public double getVraagPrijs() { return vraagprijs; }
public double getLaatsteBod() { return laatsteBod; }
public Makelaar getVerkoper() { return verkoper; }
public List<Makelaar> getGeinteresseerden() { return geinteresseerden; }
public boolean isHuidigeKoper(Makelaar makelaar) {
// Te implementeren
}
public void voegKoperToe(Makelaar makelaar) {
// Te implementeren
}
public void verwijderKoper(Makelaar makelaar) {
// Te implementeren
}
public boolean doeBod(Makelaar makelaar, double bod) {
// Te implementeren
}
```

De meeste instantievariabelen van `Pand`, de constructor en de `get`-methoden spreken voor zich. De variabele `tekoopPer` bevat de datum (`Date` object van het `java.util` package) waarop het `Pand` in de verkoop is gegaan. De variabele `laatsteBod` bevat het laatste bod van de huidige (potentiële) koper. De variabele `geinteresseerden` bevat de lijst van geinteresseerde `Makelaar`-objecten; de `Makelaar` die als eerst aan de beurt is, staat vooraan in de lijst. Merk op dat de klasse `Pand` niet over de gebruikelijke `set`-methoden beschikt om de instantievariabelen te veranderen; de meeste instantievariabelen worden alleen bij de constructie van een `Pand`-object van een waarde voorzien.

1. *(5 punten)* Implementeer de methode `gemiddeldeVraagPrijs()` van de klasse `Makelaar`.
2. *(10 punten)* Maak de specificaties van de methoden `isHuidigeKoper()`, `voegKoperToe()` en `verwijderKoper()` van de klasse `Pand` af (d.w.z., geef hun *pre- en postcondities*) en implementeer deze methoden.
3. *(5 punten)* Implementeer de methode `doeBod()` van de klasse `Makelaar`. Een makelaar kan alleen een bod doen op een pand als hij het niet zelf in de verkoop heeft. Het is niet voldoende om dit in de preconditie van `doeBod()` te eisen, d.w.z., deze conditie moet in deze methode getest worden. De methode levert `true` op als het gelukt is om een bod uit te brengen.
4. *(10 punten)* Maak de specificatie van `doeBod()` van de klasse `Pand` af (geeft *pre- en postcondities*)en implementeer de methode. Het doen van een bod op een huis is alleen toegestaan als het `Makelaar`-argument de huidige potentiële koper is. Het is niet voldoende om dit in de preconditie van `doeBod()` te eisen, d.w.z., deze conditie moet in deze methode getest worden. Verder moet een nieuw bod altijd hoger zijn dat het laatst gedane bod. De methode levert `true` op als het gelukt is om een bod uit te brengen.

Opgave 4 (30 + 5 bonus punten)

Een makelaar heeft vaak een aanzienlijk aantal panden in de verkoop. Om een goed overzicht te houden over de lijst met panden in de verkoop, moet een makelaar zijn tekoop-lijst op verschillende manieren kunnen sorteren. Daartoe wordt de volgende interface gedefiniëerd:

```
interface PandSorteerCriterium {
  public boolean kleinerDan(Pand p1, Pand p2);
}
```

1. *(5 punten)* Schrijf een klasse `PSC_TeKoopDatum` die `PandSorteerCriterium` implementeert en de methode `kleinerDan(Pand p1, Pand p2)` de waarde `true` laat opleveren als `p1` eerder in de verkoop is gegaan dan `p2`. Hiervoor kun je gebruik maken van de methode `before()` van de `Date` klasse. Deze methode heeft de volgende signatuur (van de Java 1.5 API):

   ```
   //  Tests if this date is before the specified date.
   boolean before(Date when)
   ```

2. *(5 punten)* Schrijf een klasse `PSC_AantalKopers` die `PandSorteerCriterium` implementeert en `kleinerDan(Pand p1, Pand p2)` de waarde `true` laat opleveren als `p1` meer potentiële kopers heeft dan `p2`.

3. *(5 punten + 5 bonus punten)* De volgende methode `sorteerPanden` wordt aan de klasse `Makelaar` toegevoegd om de lijst met panden in de verkoop te sorteren volgens een sorteercriterium.

   ```
   /**
    * Sorteert de lijst met Panden volgens een PandSorteerCriterium.
    * @require sc != null
    * @ensure getPanden() is gesorteerd volgens sc
    * @param sc het PandSorteerCriterium
    */
   public void sorteerPanden(PandSorteerCriterium sc)
   {
     List<Pand> pl = tekoop;
     int n = pl.size();
     // Dit algoritme is een variant op bubblesort
     for ( int i = 1; i < n; i++ ) {
       Pand p = pl.get(i);
       int j;
       for (j = i; j > 0 && sc.kleinerDan(p, pl.get(j-1)); j--)
         pl.set(j, pl.get(j-1));
       pl.set(j, p);
     }
   }
   ```

   Geef de lusinvarianten van deze methode en leg uit waarom deze lusinvarianten samen met de lusvoorwaarden de postconditie van deze methode garanderen. De lusinvariant van de binnenste lus telt als bonus punten.

4. *(15 punten)* Teken een klassendiagram met daarin de klassen `Pand`, `Makelaar`, `PandSorteerCriterium`, `PSC_TeKoopDatum` en `PSC_AantalKopers`. Hierin word je geacht om alle attributen en methoden van de klassen op te nemen (alleen hun namen).